PERCOLATION SCHEDULING WITH RESOURCE CONSTRAINTS

Kemal Ebciogiu

Alexandru Nicolau

Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No.89-31

Percolation Scheduling With Resource Constraints

Kemal Ebciogiu[1] and Alexandru Nicolau[2]

Abstract

This paper presents a new approach to resource-constrained compiler extraction of fine-grain parallelism, targeted towards VLIW supercomputers, and in particular, the IBM VLIW (Very Large Instruction Word) processor. The algorithms described integrate resource limitations into Percolation Scheduling—a global parallelization technique—to deal with resource constraints, *without* sacrificing the generality and completeness of Percolation Scheduling in the process. This is in sharp contrast with previous approaches which either applied only to conditional-free code, or drastically limited the parallelization process by imposing relatively *local* heuristic resource constraints early in the scheduling process.

1 Introduction

Automatic fine-grain (instruction level) parallelization is critical in exploiting substantially all the parallelism available in a given program, particularly highly irregular forms of parallelism not visible at coarser levels. Since the effect of all levels of parallelism exploitation have a multiplicative effect on overall performance, substantially all parallelism needs to be exploited in order to achieve good performance—an obvious consequence of Amhdal's law. The importance of fine-grain parallelism exploitation has already been recognized to some extent, and is reflected in the use of pipelining and/or horizontal microcode, in virtually all high-performance machines.

Compile-time parallelization (scheduling) of programs at the fine-grain (machine or intermediate code) level is a way to tap this level of parallelism. The obvious advantage of this approach lies in the elimination of runtime scheduling overheads (by doing the scheduling work at compile-time), and allows the utilization of sophisticated analysis and transformations that would be too expensive to perform at runtime. Furthermore, this approach can potentially exploit parallelism that is not readily available at coarser levels of granularity, and is far too tedious to be expressed at the user level.

In the last few years, parallelization techniques have emerged that can effectively *extract* such fine-grain parallelism from ordinary programs (particularly numerical codes). Unfortunately, while these techniques were able to expose parallelism, they either did not help specify how the parallelism was to be exploited on real, resource-constrained machines. While heuristics could be (and in practice have been) utilized to map the idealized schedules to machines,

[1]IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598

[2]Department of Information and Computer Science, University of California, Irvine, Ca 92717

delays and enhances performance over the previous PS programming model, where operations in an instruction were executed unconditionally.

1.2 Why use PS?

The PS transformations can expose substantial amounts of parallelism even in the presence of multiple, and statically unpredictable, conditional jumps. Furthermore, these transformations are defined independently of any superimposed heuristics, yielding increased flexibility. In [1], it was shown that the core transformations are *complete* with respect to the set of all possible local, dependency-preserving transformations on program trees. Thus, for all practical purposes, no alternate system of transformations based on the same principles (e.g., locality of application, dependency-preservation) can do better at exposing parallelism at this level.

Two additional transformations extend the applicability of the core transformations to arbitrary loops. The first of these meta-transformations allows the exploitation of fine-grain parallelism across multiple nested loops [13], while the second realizes the *full effect* of complete unwinding of loops and fine-grain parallelization, *without* the actual complete unwinding [2, 3]. The transformations work even in the presence of conditional jumps. Together, these transformations combine to overcome all of the loop-related difficulties encountered by previous fine-grain parallelizing transformations, while synthesizing the desirable parallelism-extraction features of coarser-grain transformations, such as doacross, wavefront, and loop interchange [5], [15], [4].

The resilience of our approach to statically unpredictable conditional jumps and its effectiveness has been confirmed by both our own experimental evidence [3], [12], and by independent work at IBM T.J.Watson research Center [6], [8]. In fact, Percolation Scheduling was found to be so robust in the presence of control-flow unpredictability, that its main target application in the IBM project is in systems, AI, and "casual code" domains, rather than the generally more regular and predictable numerical applications. This is in sharp contrast with previous approaches (e.g., Trace Scheduling (TS) [10]) where the parallelism extraction is either limited to a single path and/or a single rule (heuristic for path selection) is inseparable from the actual transformation mechanism.

1.3 The target resource-constrained machine: The IBM VLIW processor

A detailed description of the architecture of the IBM VLIW machine being built at IBM T.J.Watson Research Center is found in [8]. For the purposes of this paper, it is enough to describe the semantics of the execution model of the processor.

The machine can be thought of as executing *program graphs*, one node at a time. Each node in the program graph corresponds to a VLIW *instruction*, and contains a *tree* formed

In the example, L1 is the label of the instruction. If the *old* values (available from the previous instructions that were executed) of cc1 and cc2 are initially both true when L1 is executed, then the leftmost path (leading to L2) will be chosen for execution, and the operations R2:=R2+2, cc1:=R2<R5, R3:=0 will be executed (where R2<R5 will use the old value of R2, before it is incremented through R2+2); and the machine will branch to instruction labeled L2. The target instruction L2 will observe the new updated values of R2, R3 and cc1; these new values have no effect on either the selection of the path or the calculations of results in the current instruction L1. The edge emanating from the root of the VLIW instruction tree (and the operations associated with it) is called the *stub*.

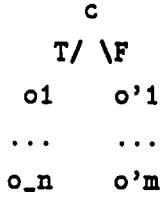
The crucial resource limitation in the IBM VLIW machine is the number of operations that can be issued in any one cycle, and the number of conditional branches in the instruction. While the approach described in this paper can be applied to other architectures, the "cleanliness" of the resource-constraints imposed by the IBM machine simplifies the task, without (by any means) making it trivial.

1.3.1 The program graph model

A program-graph consists of *nodes* that contain one or more (connected) *inverted-v*'s. The inverted-v (or iv) is the canonical form of the structures in the program graph. Initially, there is a one-to-one correspondence between nodes and iv's. Latter on, as a result of PS transformations, this correspondence may be destroyed by operations moving from one node to another. The nodes semantically correspond to IBM VLIW machine instructions, being filled through the compaction process.

An iv contains one conditional-jump (the root) and two branches (T/F), each containing zero or more operations. Each iv corresponds to a conditional test in the VLIW machine instruction along with the two (T/F) edges emanating from the test.

For example, the structure of an iv could be represented graphically as:



Definitions:
cj-branch(c,T/F) is the set of operations on c's T or F branch. *cj-target(c,T/F)* is the iv pointed to by c's T/F branch.

Since there is a one-to-one correspondence between conditional-jumps and iv's, we will refer to them interchangeably.

1.4 Previous Approaches to resource constrained scheduling.

Since even modest resource limitations make the mapping of programs to machines NP-hard, several heuristic approaches have emerged. The first and probably most widely used, is to use a set of special purpose mapping techniques, and resort to a possibly poor, but always applicable mapping, when none of the special purpose tricks apply. While this approach can yield spectacular results in some cases, its average performance is not satisfactory [10].

The second approach is to build the heuristic into the scheduling transformations. While this approach can yield better results, and is much more widely applicable if the heuristic is well chosen, it too will suffer severe drawbacks when the heuristic fails to work well. In addition the intermingling of the scheduling algorithm with the heuristic can be very unyielding to change when the code does not conform to the underlying assumption behind the heuristic, potentially yielding poor performance and rather complex code that is hard to combine with other heuristics.

PS on the other hand, manages to separate the heuristic part of the compiler from the actual transformations. This approach has been very successful so far in allowing the study of the power of the transformations without interference from the heuristics, and allows maximum flexibility in the use of heuristic scheduling techniques where and when needed. The question that we are faced with is how to introduce resource constraints without violating this separation between transformations and heuristics, and without undully restricting the transformation process. The integration of heuristics designed to manage resources in the scheduling algorithm tends to limit the global effect of the parallelization algorithm, by forcing it into early decisions as to what should be moved and where. Most of the obvious heuristics that could be used in this context lead to "myopic" code-generation, due largely to the difficulty to maintain truly global information from which the next best transformation can be chosen.

The alternate approach, taken by Trace Scheduling[3] and previous PS versions, has been to delay all resource mapping till the last possible time, and first generate idealized schedules that do not take resources into account. These schedules, being only restricted by the ability of the transformations to extract parallelism, can then be mapped heuristically onto a given machine. The drawback of this approach is that it is very hard to maintain a global view of the code while doing the mapping, particularly across potential execution paths through the code While this approach often does well, it has the distinct drawback that it may have to undo some of the code motions performed in the unrestricted phase. This is complicated by the fact that some transformations do not have a unique inverse, and undoing them may generate different (and potentially worse) code than that of the original program. An additional

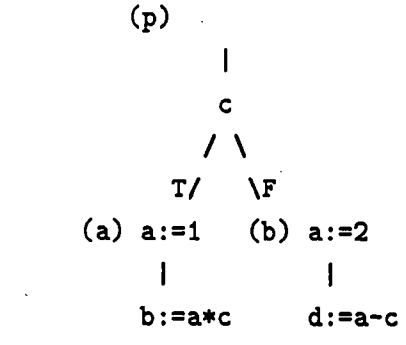
[3]Trace Scheduling has some built-in heuristics for picking and prioritizing "traces" to be parallelized, based on some heuristic estimate of the direction conditional-jumps will take at runtime. While this tends to restrict the parallelism extraction capabilities of TS vis a vis PS, it is not dealing with resources per se.

2.1 What we're proposing

To achieve the goals just mentioned, we need to know, for every machine instruction, what operations can *potentially* be scheduled in it. Then we could use the full power of PS to transform the code, allowing only the most important k operations to move into that instruction, where k is the maximal number of resources available.[5] This framework is analogous to that of classical global optimizations (e.g., constant-folding) with the initial computation of "operations that can be scheduled in instruction k" (*unifiable-ops*) corresponding to data-flow (reaching) information, and the ensuing percolation scheduling phase corresponding to the optimization itself. In this framework, the heuristics only enter in the choice of the order in which the instructions (and operation within them) are processed.

2.2 The algorithm

The algorithm consists of two phases. First, the information as to what operations can move and where, is gathered. In the second phase, the actual scheduling takes place through PS transformations. This in turn will necessitate a continuous (incremental) updating of the original information, to reflect the changing of the program. The gathering of information in this method, while akin to classical flow-analysis, is less conservative. The information that we compile is the set of *all* operations that may, as a result of PS parallelizing transformations, reach a certain point in the program, even though some of these motions may be mutually exclusive. For example, consider the situation bellow:



Either operation (a) or (b) may move to point p, but not both. To allow maximum flexibility in scheduling, the set of operations that can potentially reach point p should initially include both (a) and (b). We will refer to operations in this set as *mutually blocking or mutually*

[5]The heuristic part of our algorithm is a generalization of List-scheduling's estimate function based on dependency chains, that accommodates the need to integrate the weight of multiple paths through the code. In addition, since code motions in the presence of complex control flow are not as trivial as in straight-line code, we need to maintain information as to the set of operations that may move, through semantics preserving transformations, at any given point in the code, as well as the impact of these transformations on the rest of the program. Computing and maintaining this information on movable operations is the crux of our algorithm.

each node. This however, makes the (incremental) updating of this information slightly less efficient when nodes grow to contain more than one iv.

Definitions:

A *path* **in** a node n to a successor node s is the sequence of operations and directions (for conditional branches) by which node s can be reached at execution time from the top of node n. Note that there may be several paths from n to s. We will denote by $paths(n, s)$ the set of paths from the top of node n to node s. A sequence formed by a path **in** node n_0 (to n_1), followed by a path in n_1 to n_2, etc, will also be called a *path*, the difference between the two being clear from context. The edge emanating from the last iv on a path within a node is called the *tip* of that path.

An operation o is said to *reach* a node n, if there exists a path from o's current node to n along which no data-dependencies that could prevent o from moving exist.

The following definition describes the set of operations in the program that can reach c's iv and that are dependent on an operation in that branch of c:

Killed-with-live-ops(cj-branch(c,dir)) = killed-with-live-ops(c,dir) $=$
$\{o \| (o \in Program\ ) \bigwedge(((\exists o' \neq o \in cj\text{-}branch(c, dir)\ s.t.\ o\ dep\ o')) \bigvee((live\text{-}vars(c, \overline{dir}) \cap write(o)) \neq \emptyset))\}$

where *dir* is either *true* (T) or *false* (F), and *dep* is one of:

- write-before-read
- read-before-write
- write-of-live-at-branch

Note that *dep* should be a true dependency in the sense that the operations should have occurred in such an order as to make the dependency actually occur; this would not be automatically the case if the sets are implemented as bit vectors, but can easily be enforced by proper numbering so that for all o, o' such that $o\ dep\ o'$, $(\#bit(o') < \#bit(o))$. Also note that write-before-write is not a dependency. This is due to the fact that the execution model has a well defined mechanism for dealing with multiple writes within a node, and thus there is no reason to disallow them.[6]

The following is the definition of the set of operations not allowed to move above c, only because they write a variable being read on (another) branch of c.
Killed-by-live-only(c,dir) $= \{o | (o \in Program) \bigwedge((live-vars(c, \overline{dir}) \cap write(o)) \neq \emptyset)\}$.

Note that *killed-with-live-ops(c,dir)* can and should correctly include operations in the *cj-branch(c,dir)* itself. To correctly do this in a bit-vector implementation, operations in the

[6] In fact, if each write reaches a potential use—on separate paths—then allowing the occurrence of two such writes in the same node may benefit (speed-up) execution; otherwise, dead writes can be removed as part of the PS transformations.

Set5 = unifiable-ops(c,T) ∩ killed-by-live-only(c,T).

Set6 = unifiable-ops(c,F) ∩ killed-by-live-only(c,F).

Set7 = cj-branch(c,T) ∩ Killed-by-live-only(c,T).

Set8 = cj-branch(c,F) ∩ Killed-by-live-only(c,F).

Set9 = { c }.

The computation of the *unifiable-ops* set can be further simplified, by using the commutativity of the set-union operator, as well as the fact that $(A \wedge B) \vee (C \wedge B) = (A \vee C) \wedge B$. This yields the following simplification:

Let set U(c,dir) = (unifiable-ops(c,dir) ∪ cj-branch(c,dir)), K(c,dir) = killed-with-live-ops(c,dir), and L(c,dir) = killed-by-live-only(c,dir).

then

Set1 ∪ Set3 = U(c,T) - K(c,T).

Set2 ∪ Set4= U(c,F) - K(c,F).

Set5 ∪ Set7 = U(c,T) ∩ L(c,T).

Set6 ∪ Set8 = U(c,F) ∩ L(c,F).

Thus

unifiable-ops(c) = $(U(c,T) \cap \overline{K(c,T)}) \cup (U(c,F) \cap \overline{K(c,F)}) \cup [(U(c,T) \cap L(c,T)) \cap (U(c,F) \cap L(c,F))] \cup \{c\}$,

which may be simplified further.

2.2.2 Performing Resource-constrained PS

The scheduling with resources relies on Percolation Scheduling core transformations (to be defined shortly), based on [11], [12], [6]. The main procedure *schedule-with-resources* applies to *nodes* in a heuristically-defined order (an optimal solution to the scheduling with resource being NP-hard). The procedure calls an auxiliary routine, migrate, that brings the operation being moved up to, but not into the current node being filled. The filled nodes map directly into instructions for the IBM VLIW machine. The order in which the scheduling occurs is controlled by three routines, *choose-op, choose-branch*, and *choose-node*. Useful heuristics for these routines are discussed in the next section.

The following procedures are expressed in terms of nodes, rather than iv's for convenience of explanation, as the PS transformations conceptually apply to program-graph nodes.

```
PROCEDURE schedule-with-resources (n:node) /* n is a node in the program
                                              graph */
WHILE there exist unfilled nodes, n, DO
 choose-node n;

 WHILE (resources not full) DO
```

2.2.3 Incremental Updating of Unifiable-ops

Fortunately for the efficiency—and hence the practical applicability—of the algorithm, the updating of *unifiable-ops* sets for each node, can become a local operation, affecting only the nodes actually transformed by a PS core transformation in the process of *scheduling-with-resources*. There are only two requirements for this incremental process to be correct. First, all predecessors of a node should be scheduled before the node is scheduled. If this condition is not satisfied, unifiable-ops sets for some nodes may contain *more* operations than can actually be moved up to that point (they can never contain less). Second, a slightly more complex splitting scheme is needed to keep paths sharing a common section of the program graph from resulting in incorrect unifiable-ops information in the presence of mutually blocking operations.[7] When an operation that is causing a mutually-blocking conflict moves out of a node[8], the copy node is *always* preserved (even though there may not appear to be multiple paths through the original node). This process then continues along the path on which the operation is being moved, copy nodes always being preserved and threaded into an alternate path. If points where the nodes would normally be split (by regular PS) are encountered, then the split occurs as usual (see the description of the move transformations) but the copy node is threaded into the alternate, rather than the new, path (i.e., not into the path on which the blockage occurs due to the operation having moved). If no such points are encountered until the operation's final position, then the alternate path (formed by the copy nodes) can be discarded. This different split mechanism would be required to avoid the problem illustrated in Figure 1. Initially, either $a := 1$, or $a := 2$ can move to any of the nodes A, B, , or G, and thus would be part of the *unifiable-ops* sets for these nodes. Moving $a := 1$ from node E to node A, however, leaves a use of a exposed, on paths to B and G, so that $a := 2$ would not be able to move to either B or G. Thus, unless *unifiable-ops* sets are updated for these nodes, they will contain incorrect information. However, since B and G are not touched by the migration of $a := 1$ to A, this would require (in general) an extensive recomputation of *unifiable-ops*, throughout the program graph. The split modification described above, fixes this problem by ensuring that paths get split early enough (in this case at D) so that consistent *unifiable-ops* information is maintained.

Since the PS transformations could (would) check applicability before transforming the program, only efficiency and not correctness could be affected even if unifiable-ops were not updated. Furthermore, even restoring precise unifiable-ops sets in the presence of arbitrary-

[7] *Splitting* is the duplication of a node that occurs in PS to maintain the semantic correctness on paths through that node that are distinct from the path on which the transformation occurs (see the definition of the PS transformations bellow).

[8] This can be easily detected on the fly: unifiable-ops for the node will normally only loose the operation being moved; whenever other operations are removed from unifiable-ops for the node as a result of a single operation moving out of it, this indicates that the operation being moved is a mutually blocking operation.

on some path from n, are read before being written.

```
PROCEDURE move-op-or-cj(n',n,tp,op) /* also does updating of
                                       unifiable-ops */


  IF op is not a conditional-jump THEN
    /* move all syntactic copies of op from
       node n' to the end of path tp of n (which leads to n')  */

     move-op(n', n, tp, op);       /* the PS transformation with live
                                      information updating */
     IF move was successful THEN
       /*  n'' is a new node created by move-op, (a copy of n' with all
           syntactic copies of o deleted */
        update-live-dead(n'');
        update-unifiable-ops(n'');
     ENDIF
  ELSE
     move-cj(n',n,tp,op);        /* the PS transformation with live
                                    information updating */
     IF move was successful THEN
       /* nT', nF' are two partial copies of n', for the T/F branch of op */
        update-live-dead(nT');
        update-unifiable-ops(nT');
        update-live-dead(nF');
        update-unifiable-ops(nF');
     ENDIF
  ENDIF
END move-op-or-cj

PROCEDURE update-unifiable-ops(n:node);
 FOR i in (iv's in node n) DO /* this restrains compute-unifiable-ops
                                 from recomputing beyond the current node */
   marked(i) := unvisited;

 compute-unifiable-ops(n);
END update-unifiable-ops;
```

The procedure *compute-unifiable-ops* can be used in $update-unifiable-ops$ eventhough

PS transformations below apply to nodes, containing tree-like structures, (constructed out of the iv's described earlier.

```
Procedure Move-op(o:operation; n:from-node; m:to-node; p:path-for-move);

IF      /* no conflict in m on relevant paths */
      for every o' on path p in m, from m to n,
          intersection(reads(o),writes(o')) = nil
    AND  /* no ops in n read what o writes */
      for every o' in n, o' =/= o,
          intersection(writes(o),reads(o')) = nil
    AND  /* either writes(o) is dead on all paths FROM n, or it's killed in n*/
      for every s in succs(n), either
          intersection(live-at-top(s),writes(o)) = nil
        OR
          for every path, p', from n to s, there exists an operation o' in p',
            s.t.
             writes(o) = writes(o')
THEN /* move */
    create a copy n' of n;
    delete all occurrences of o in n'; /* unification */
    move o into tip of path p in m; /*add o to last cj-branch on m on path p*/
    make m go to n' instead of n (on path p ONLY);
    IF there exists o' s.t. writes(o') = writes(o)
      AND
       o' occurs on a path in m going through the newly added o,
    THEN
      delete o' from these paths;
      for all iv's j, occurring on paths encompassing o, and
              the original location of o',
         place o' in cj-branch(j,dir), where dir is the branch
                not leading to o;
         if o' becomes dead, remove it altogether;
/*  n is the split copy of itself; n' is the transformed version */
    IF n has no predecessors, delete it;
   END /* move */
END /* move-op */
```

```
    and make:
      cj-target(iv(o),T) := nT;
      cj-target(iv(o),F) := nF.
    Thus, the last iv in m,
        |op1...
        |opi
        n
     becomes
        |op1 ...
        |opi
     o:if cj       /* new iv rooted in o */
      /   \
    nT     nF
  /* n is the split copy of itself;(nT and nF are the transformed versions)*/
   If n now has no predecessors delete it from the program.
  END /* then */
```

To re-compute live variables of a node n after move-op or move-cj, we can use the following scheme:

```
procedure update-live-vars(n:node)
 let writes(p:path) be the set of registers assigned to along path p.
 let reads(p:path)  be the set of registers read along path p.

    livevars(n) := nil;
    FOR each path p from n to {s| s in succs(n)} DO
        livevars(n) := Union(livevars(n), Union([livevars(s) - writes(p)],
                                                reads(p)));
END update-live-vars;
```

2.4 Choosing Heuristics

One of the advantages of our approach is that the heuristics can be chosen independently of the actual code transformations, and even of the *selection of the candidates* for motions from among which the choices are made (this selection is *not* heuristic). Furthermore, [1] has formally shown that the code transformations achievable by applications of PS are for all practical purposes complete, i.e., that their expressive power is as great as can be expected from local, dependency preserving transformations.

a PS transformation) updated, without loss of correctness or accuracy.

Proof: As a result of PS motions, uses (reads) of variables may become exposed (live) at places where they were not. Similarly, when the last use of a variable is moved upward, the variable will be dead in nodes where it used to be live. Thus updating *is* necessary. Fortunately, the only place where this effect will manifest itself is on paths in the node from which an operation has just moved. Thus recomputing the live information for that node (by propagating it once from the targets of the node to its top) will restore its correctness. PS transformations ensure that no operations ever move out of a path, nor move into a path if they can affect live information on that path [1]. Thus the live information cannot change globally.)□

Theorem 1: Updating of unifiable-ops is a necessary and incremental process, when the order of traversal of the nodes and the splitting mechanism in PS are as previously discussed.

Proof: Unifiable-ops changes, by definition, as a result of PS transformations. As long as all predecessors of a node are processed before the node itself is processed, there are only three possible situations that may *potentially* cause changes to unifiable-ops. We will examine each one in turn, and show that all of them either cause no change to unifiable-ops, or that the changes can be incorporated incrementally.

- Since operations cannot move out of paths as a result of PS, unifiable-ops for a node cannot change (i.e., operations will not have to be subtracted from unifiable-ops) due to other operations disappearing from way under that node. (Of course, unifiable-ops for a node *can* change locally by having an operation move out of the node, but that can be updated locally.) The only other case where an operation may need to be subtracted from unifiable-ops is when one of two mutually blocking operations moves past the iv, and thus prevents the alternate(s) from moving. Note that (given the definition of move-op) this will only happen when the operation actually moves *out* of the node containing the iv causing the blocking. If nodes are processed in predecessors-first order as discussed earlier, then all the nodes *on the motion's path* affected by this change to unifiable-ops can indeed be updated (incrementally) as a result of the *migrate* (PS) process. The modified splitting mechanism described in Section 2.2.3 will preserve the original nodes (with all alternate move options) for all other paths, and thus unifiable-ops for nodes on these paths are not affected.

- Operations may move into a path as a result of PS transformations, but then all such operations are already accounted for in the initial unifiable-ops computation, (which for every node is the union over all operations on all possible paths from the node that can move to the top of the node). Thus no change (addition) to unifiable-ops is necessary.

- Operations that are originally not in unifiable-ops, because they are "masked" by other operations in unifiable-ops on which they depend, may become eligible for unifiable-ops

each path reduces to the case above.

- Code with conditional jumps. Operations can prevent other operations from moving past conditionals. But this is precisely the (only) situation where a choice is required even in PS without resource constraints. If the choice is the same, then the resource constrained PS will produce identical schedules. □

Theorem 5: Subject to the data-dependencies and the ordering imposed by the *choice* functions, PS-with-resources is optimal. That is, for each node that can contain k ops, (picking the nodes in choice order), the k most important operations (based on choice (and availability) at the time the node is scheduled) are indeed scheduled in that node.

Proof: By the previous theorem, when resources are plentiful relative to the requirements of the code and there is no conditional flow-of-control the operations are scheduled as early as their dependencies allow, which is clearly optimal. Furthermore, if we assume that the choice heuristics for prioritizing paths, nodes, and operations *are* optimal, then the resulting schedule for the first path should be optimal. However, a problem arises in analyzing the schedule of the next and following paths, as operations may have moved into a node (which was originally shared) as a byproduct of the operation being moved on a more important path. The question that arises is whether—subject to our choice heuristics when applied to this new path—the operation motion is indeed optimal *for all paths involved.* While with an arbitrary set of heuristics this may not be the case, our heuristics do not give rise to such inconsistencies. Indeed, when an operation is moved to a node, even as a side-effect of motions on another path, that motion is also the best available heuristic choice for all the other paths sharing that node (since the operation occurs on the shared portion of several paths, its probability of execution is greater than any operations on the non-shared portions of the paths thus for any of the paths sharing that common portion, the operation chosen (by the evaluation-function for choose-op) will be the same, as the probability and the dependents factor in the evaluation function for choose-ops are the same. Since the operation was chosen subject to this metric as the most important candidate to move at this point on the currently most important path, it will also be the most important candidate to move on any of the alternate paths. Thus it will either move as high as the split node—the last common node—or higher on any of these paths. In other words, the robustness of the heuristics guarantees that operations with highest priority (according to the heuristics) are scheduled earliest, *on all paths.* The only remaining problem is that multiple paths going through a node, could result ultimately in the overloading of that node, due to repeated splitting, as illustrated in the situation bellow.

4 An example of how resource constrained PS works

The following is a simple demonstration of our algorithm. Let us assume that we have an IBM VLIW style machine with a maximum of 2 arithmetic or load/store ops and a maximum of two way branching per instruction. Assume the following is the input sequential code, where each node consists of a single iv. The initial $unifiable_ops$ set of each node is shown above the node. The instructions below are each intended to encode a VLIW tree, we hope that the notation is obvious.

```
/* find the minimum of an array */
/* t:=(ai) means use ai as an address register and
 fetch the word at address ai into register t  */

   {cc0:=ai<ailim, t:=(ai), ai:=ai+4}     /* unifiable-ops set for node 1*/
loop: cc0:=ai<ailim, goto L2              /* node 1 */

    {not cc0, t:=(ai), ai:=ai+4}          /* unifiable-ops set for node 2*/
L2: if not cc0 (goto exit) else (goto L3) /* node 2 */

    {t:=(ai), ai:=ai+4}                   /* unifiable-ops set for node 3*/
L3: t:=(ai), goto L4                      /* node 3 */

{cc1:=t<min, ai:=ai+4}                    /* unifiable-ops set for node 4*/
L4: cc1:=t<min, goto L5                   /* node 4 */

{not cc1, ai:=ai+4}                       /* unifiable-ops set for node 5*/
L5: if not cc1 (goto X1) else (goto L6)   /* node 5 */

{min:=t, ai:=ai+4}                        /* unifiable-ops set for node 6*/
L6: min:=t, goto X1                       /* node 6 */

{ai:=ai+4}                                /* unifiable-ops set for node 7/
X1: ai:=ai+4, goto loop                   /* node 7 */

exit: t is dead here
```

Suppose that the ALU ops and tests that are control-dependent on a test (conditional branch) are counted among the "number of dependents" of the test for the purposes of *choose-op*.

Scheduling will start, guided by choose-node, with the first node. To fill this node, the candidates in unifiable-ops are considered in the order dictated by choose-op:

The final result, after filling node L5' will be:

```
Eligible ops for moving to L5':
1st: min:=t: 0 dep 20% (guess for probability)

loop: cc0:=ai<ailim, t:=(ai), goto L2'
L2':  if not cc0 (goto exit) else (cc1:=t<min, ai:=ai+4, goto L5')
L5':  if not cc1 (goto loop) else (min:=t, goto loop)
```

Thus the execution of the loop was compressed to only 3 cycles/iteration; this is optimal assuming no software pipelining or unrolling of the loop.

5 Conclusions

In the spirit of PS, our approach to resource-constrained scheduling is highly modular; other heuristics than the ones described can be introduced, by simply changing the choice procedures, the only requirement (for efficiency of updating) being that nodes be traversed in predecessors-first order.

We have presented the first resource-constrained fine-grain parallelization algorithm to apply uniformly on multiple paths. This is in contrast to list scheduling, which strictly speaking applies only to basic blocks, or Trace Scheduling which only considers one path at a time. Without additional techniques such as software pipelining and/or loop unrolling, the technique described in this paper will of course not produce the highest possible parallelism, but it can serve as a crucial component of a fine-grain parallelizing compiler, and is important for this reason. We believe that this work contributes yet another step on the way to the efficient automatic parallelization of ordinary code, *for all applications.*

References

[1] A.Aiken. *Compaction Based Parallelization.* Ph.D. Thesis, Department of Computer Science, Cornell University, August 1988.

[2] A.Aiken and A.Nicolau. *Perfect Pipelining: A New Loop Parallelization Technique.* Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988.

[3] A.Aiken and A.Nicolau. *Optimal Loop Parallelization*, Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1988.